

Original citation:

Alexander-Craig, I. D. (1988) WINNSOME : a neural network simulation package. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-120

Permanent WRAP url:

<http://wrap.warwick.ac.uk/60816>

Copyright and reuse:

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:

The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: publications@warwick.ac.uk



<http://wrap.warwick.ac.uk/>

Research Report 120

WINNSOME: A NEURAL NETWORK SIMULATION PACKAGE

Iain D Craig

(RR120)

This paper describes a neural network simulation package that has some novel features. The package is designed to be reasonably fast at runtime and contains facilities for user-specification of unit and connection types. A new structuring concept, the module, is introduced by the package. The module construct is designed to allow model builders construct their simulations in a modular fashion, and is also included so that functionally distinct regions can be defined and their interactions regularised. The module concept is justified in terms of neuroanatomy.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

March 1988

WINNSOME: A NEURAL NETWORK SIMULATION PACKAGE

Iain D. Craig

Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK

ABSTRACT

This paper describes a neural network simulation package that has some novel features. The package is designed to be reasonably fast at runtime and contains facilities for user-specification of unit and connection types. A new structuring concept, the module, is introduced by the package. The module construct is designed to allow model builders construct their simulations in a modular fashion, and is also included so that functionally distinct regions can be defined and their interactions regularised. The module concept is justified in terms of neuroanatomy.

1. INTRODUCTION

Despite the fact that research into neural networks is a growth area, a great deal of the software needed to simulate these structures remains either *ad hoc* or else computationally extremely expensive even though it is rather simple, structurally. Computational cost is unlikely to reduce significantly because neural network simulations involve large numbers of operations over large numbers of basic computing elements. In addition, activation has to be passed along connections between units and this, too, can be expensive. Finally, there is the point that neural simulations currently serialise what is naturally a parallel processing system: serialisation is expensive in terms of time and memory.

Despite these problems, it is possible to avoid some of the problems associated with this kind of software. This paper describes a network simulation package

called WINNSOME¹ which is currently under development in the C++ programming language (Soustrup, 1986) to ensure maximum portability. C++ also has the advantages that the code produced by it can be optimized and that storage allocation can be made the responsibility of the application programmer (the WINNSOME package in the present case) so that expensive, general purpose solutions can be avoided. This contrasts with, for example, Zipser's (1986) P3 simulator (which is written in ZetaLISP and runs on Symbolics hardware): WINNSOME runs on any system that supports C++.

WINNSOME introduces some new concepts into neural network modeling and is designed to allow users to define their own unit and connection types. The package contains constructs that allow networks to be defined in a modular fashion without restricting inter-unit connections. The primary structuring device provided by WINNSOME is the *module* construct. A module is a collection of processing elements (units) that together perform some function. Each module can be thought of as implementing at least one function (e.g., discriminating between bilabial plosives). The intention behind the module construct is that it forms a basic architectural unit with which to construct simulations: in this respect, it is analogous to a cortical macro-column (Shepherd, 1979; Squire, 1987) or to an entire region (e.g., the olfactory bulb, Skarda, 1987).

Modules can be connected together in a variety of ways providing an additional level of structure over and above that provided by simple collections of inter-connected units. As will be seen, modules allow for connections to be made in ways other than simply allowing the output of one module to become the input to another. This property of the module construct allows inter-module connections that resemble projections from deeper brain regions to layers in the neocortex.

¹ WINNSOME stands for WarWick Neural Network SimulatiOn Environment. The name is also connected to a basic fact about research grants: you win some, you loose some - usually the latter.

In addition to the module construct, the WINNSOME package allows the user to define unit and connection types which are specific to the structures being modelled or to the problem under attack. The package also allows users to combine different unit and/or connection types within the same module. This facility would be required, for example, when modelling the neocortex - it would allow pyramidal cells to be incorporated into the same module as granule cells.

The structure of this paper is as follows. In the next section, the concepts of unit and connection are introduced and interpreted within the context of an object-oriented programming language. In section three the module construct is introduced and justified; the section also contains examples of various interconnection schemes between modules as well as explaining the various basic module types and the ways in which their units can be connected together. Section four summarises the main points and contains some pointers to future developments.

Acknowledgements

I would like to thank Mark Rafter for helping me understand the C++ language well enough to build the package. Some of the requirements connected with visual processing were suggested by Roland Wilson and David Randell. All errors and misinterpretations are, of course, my own.

2. UNITS AND CONNECTIONS

In this section, the concepts of *unit* and *connection* are explained, first in general terms and then within the context of the WINNSOME simulator.

2.1 Units and Connections in general

Neural network models are composed of a set of units which are linked together by connections to form a network. Each unit represents an item of information or a structure in some domain (for example, a unit might represent a phoneme, say

/p/).

A unit can be thought of as having a state, an input function and an output function. The *state* can be a binary value or can take discrete values within a range or it can be continuous. The state is usually taken to represent the activation level of a unit: information in neural network models is represented in terms of activation levels, both locally and globally. Activation is passed between units along connections.

When a unit is given external excitation (activation), its *input function* transforms the external value and updates the unit's internal state. Similarly, the *output function* transforms a unit's internal state into some form that can be used by other units. Input functions can also be used to provide units with threshold operations: such units only become active when the net input they receive is above some threshold value.

Units are linked together via *connections*. A connection joins two units and acts as the medium through which activation is passed between units. Each connection is usually associated with a *weight*. The weight represents the strength of the connection between the units it joins. The weight is used to calculate the activation value for a unit. This is typically done as follows: for each unit, multiply the activation values of all the units from which it receives input by the weight of the connections joining them to the unit.

When units are connected, the activation of one can affect the activation of the other - this is precisely the function outlined above. The activation change can happen, basically, in one of two ways - the connection can be inhibitory or excitatory. If a connection is inhibitory, the activation of one unit is used to suppress (reduce) the activation of the other. If, on the other hand, a connection is excitatory, the activation of one unit increases the activation of the one to which it is connected. For example, an inhibitory connection might be represented by a negative weight, and an excitatory one by a positive value.

The relationship between units is also expressed by the form that their

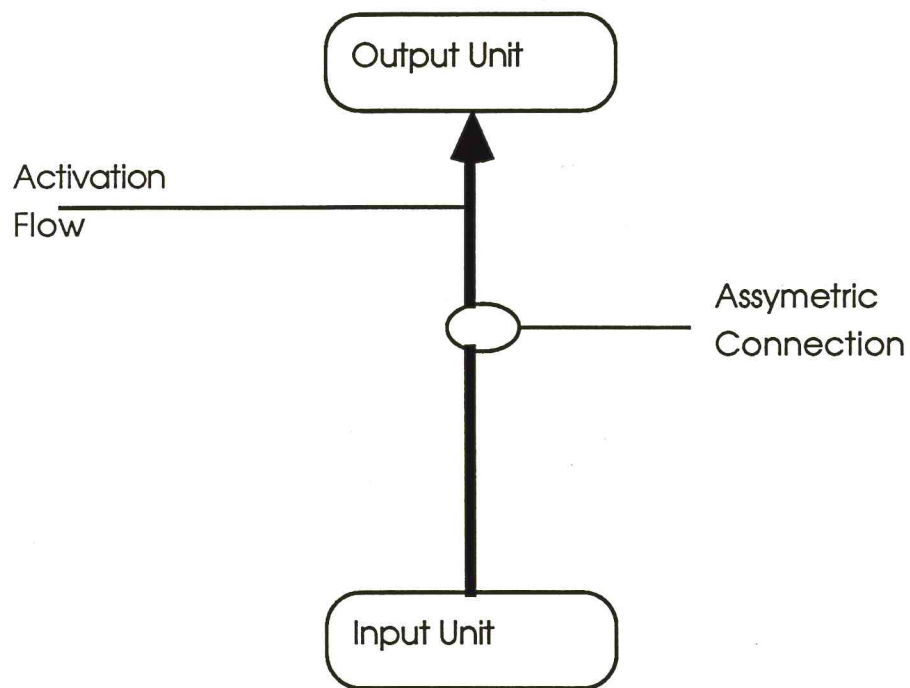
connections have: connections can be asymmetric (uni-directional) or symmetric (bi-directional). Asymmetric connections allow activation to be passed in only one direction; symmetric connections allow activation to be passed in both directions. If, for example, there are two units U_1 and U_2 , and the connection between them is asymmetric and has the form $\langle U_1, U_2 \rangle$, activation can only be passed from U_1 to U_2 and not in the reverse direction. If the connection is symmetric, activation can be passed from U_1 to U_2 and from U_2 to U_1 . Both asymmetric and symmetric connections are shown in Figure 1.

2.2 Units and Connections in WINNSOME

WINNSOME simulations are composed of units that are connected in various ways to form networks. Although it is expected that most simulations will be built using units and connections of the form described above, it is still possible that highly specific units and connections will have to be defined in some cases.

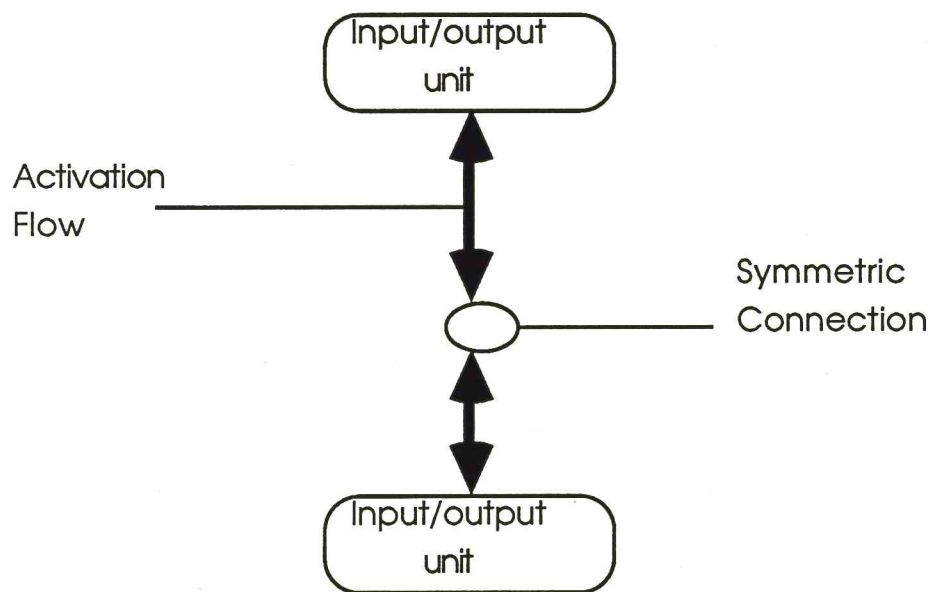
Even given the structures described in the general case above, there is still room for negotiation. For example, the input and output functions have to be defined for each unit. Some unit structure details vary according to the learning rule used (assuming that the network engages in learning and is not pre-structured). Units and connections for the Generalised Delta Rule (Rumelhart, 1986) are very slightly different from those for the Simple Delta Rule and both of these rules require that connections have a delta value stored in them for use in updating the connection strength during learning. Finally, most networks operate by propagating activation from one unit to another. The propagation rule can vary between models so that, for example, in one model, propagation may be a function that is implemented uniformly across an entire network; in another model, each unit might have its own propagation operation. In another, the propagation operation might be defined for each connection type: such a scheme might be closer to a model of actual synapses in some region of the brain.

WINNSOME has been designed so that model builders can define exactly the



(a)

Assymmetric connection



(b)

Symmtric Connection

Figure 1 Connection types

type of network they require. Network definition involves specifying the structure of units and connections. The package provides some basic unit and connection types (e.g., a standard unit, a unit suitable for the Delta Rule and a unit and connection type suitable for the Generalised Delta Rule), but it also allows users to define their own types if, and when, the need arises. For example, if propagation is to be handled by individual units, this would be done by defining a new unit type.

In order to make the definition of new unit and connection types as easy as possible, facilities provided by the implementation language are exploited. C++ is an object-oriented programming language which supports inheritance. The basic unit types are implemented as objects in C++ and they inherit slots and methods from a common set of base objects. Thus, many of the features and facilities needed to implement a unit or connection type are already present in the simulation software (and have to be present so that the basic types can be implemented). The basic unit type, for example, contains a propagation operation that can be used by any other unit: if the application requires propagation to be performed on a network-wide basis, this operation is simply ignored.

In order to make the interfaces between package- and user-defined unit and connection types as regular as possible, a number of conventions and definitions have been adopted. Activation values can, in reality, be of any type one might wish (binary, integer, discrete range or continuous), but WINNSOME provides a basic activation value type which is to be used for all cases in which activation has to be represented. The activation-value type makes value-passing inside networks more uniform. More will be said on activation values in the next section.

So that activation can be passed to, activation obtained from a unit, or propagated along a connection, WINNSOME provides standard methods which are to be used in all cases. For example, to obtain the output from a unit, the unit's `UNIT_OUTPUT` method is called. Similarly, to set the activation level of a connection, the connection's `SET_CONNECTION_STRENGTH` method is used. Both

of these methods manipulate objects of the activation value type, so a uniform protocol is supported. Because units and connections are implemented as objects, these methods (and others besides) are defined for each individual instance of the class, so to set the input of any unit, the instance of `SET_INPUT` found on the unit itself is executed: again, this makes the definition of new unit and connection types somewhat simpler than would otherwise be the case. It should be noted that the basic connection type provided by the package is asymmetric: this is because most naturally occurring synapses have this property.

One of the design goals of WINNSOME was that networks containing units of different types should be possible. In other words, it should be possible to construct a network that contains units with different behaviours and which are linked by connections that implement different functions. The reason for this is that, for example, the neocortex contains cells of different types and they are not separated into regions according to cell type. In some implementation languages, this would have been a hard task and it might have been possible to implement only the most general unit and connection types within the package itself.

Because WINNSOME is implemented in C++, it was possible to define *basic* unit and connection types which provide some of the features that might be necessary: the remaining features are defined by the model builder on a simulation-specific basis. There are no type-checking problems generated by the requirement that heterogeneous networks be possible because all units and connections are derived from a single basic type - one for units and one for connections. Thus any unit and any connection can be used in place of any other without causing type-checking problems. This property is particularly desirable when modules are constructed (see below) because it means that a module can contain units of any type whatsoever.

If the protocols defined for units and connections are observed, there should be, in general, no need for code within a simulation to have information about which type of unit or connection it is manipulating at any time. Thus, there

should be no need to employ type-discriminating code within simulations in order to determine what should be done with any given unit. This leads to uniform treatment of units and a simpler simulation structure; it also reduces the overhead involved in running networks because additional operations such as type-discrimination are obviated.

For example, each unit is equipped with an input function, a state-setting function and an output function. The external interface to a unit is expressed in terms of the `SET_INPUT` and `UNIT_OUTPUT` methods. What happens for any instance of a unit type when these methods are invoked is a matter that concerns only the unit type itself and not external software constructed only for control. Under this régime, even propagation can be performed automatically when the input and output methods are invoked, although, as has been noted, this depends upon the specifics of any given model.

The facilities defined for the basic unit and connection types are inherited by all new types defined as extensions to them. It is, therefore, possible for new unit and connection types to be defined and used in simulations without the need to construct additional support software.

3. MODULES

Modules are the basic large-grain structuring device provided by WINNSOME. They collect related units together to form larger-scale components from which to construct simulations.

A module may collect units on the basis of function (e.g., that the units within the module are together necessary to compute the function represented by the module) or on the basis of structure (e.g., a set of modules, each containing units of uniform type, might be constructed so that the same pattern can be learned in different ways by each individual module, thus providing facilities to perform competitive learning, pattern completion and auto-association within the same simulation).

Modules can be connected together in various ways depending upon how they are to be used and how their inputs are derived.

3.1 Basic Module Types

Modules divide up according to their internal structure and according to the kinds of inputs they can receive. Any kind of unit can be located within a module: the module construct is completely general and was designed so that units of different types could be located together. The basic package supports three kinds of general module structure:

- Modules with a single layer of units (it is quite possible for a module of this sort to contain only a single unit);
- Modules with an input and an output layer;
- Modules with an input and an output layer and one or more layers of hidden units.

In the case of modules that have specific and separate input and output layers, the number of units in the input layer need not be the same as the number of output units. Also, when a module contains hidden units, the number of units in a layer need not be the same as the number of units in any other layer. In other words, if one thinks of the layers (input, output and hidden) in a module as a matrix, it need not be square. These three basic module types are depicted in Figures 2, 3 and 4.

This basic classification is extended by WINNSOME to allow modules to be composed of layers that are either vectors (one-dimensional arrays) or surfaces (two-dimensional arrays) of units. Surfaces are clearly of interest when the problem under attack requires two-dimensional input or output, as is the case in visual processing. Indeed, the idea of surfaces was suggested by the need to be able to process images in WINNSOME networks: each unit in a surface can be thought of as something analogous to a pixel or to a neuron in a retinocentric structure.

The package also supports a type called a *volume*. A volume is a three-

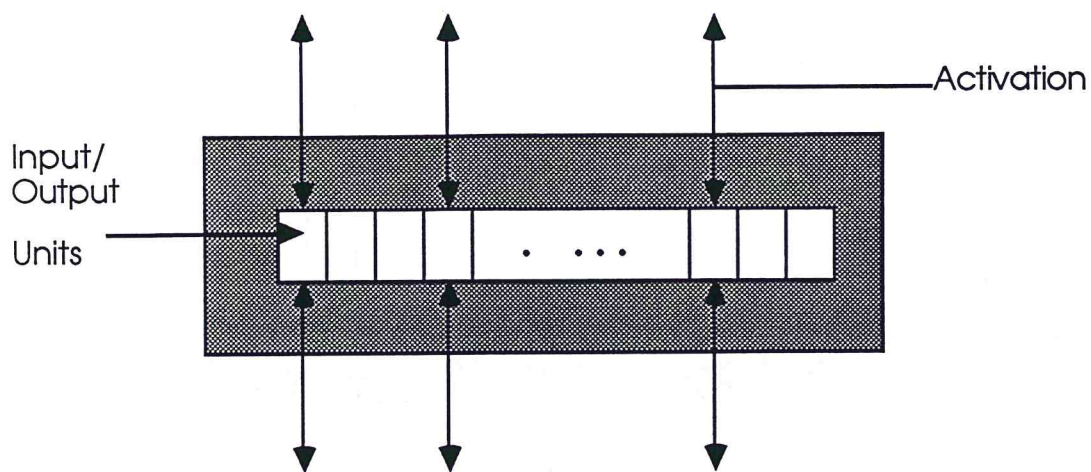


Figure 2 A Single Layer Module

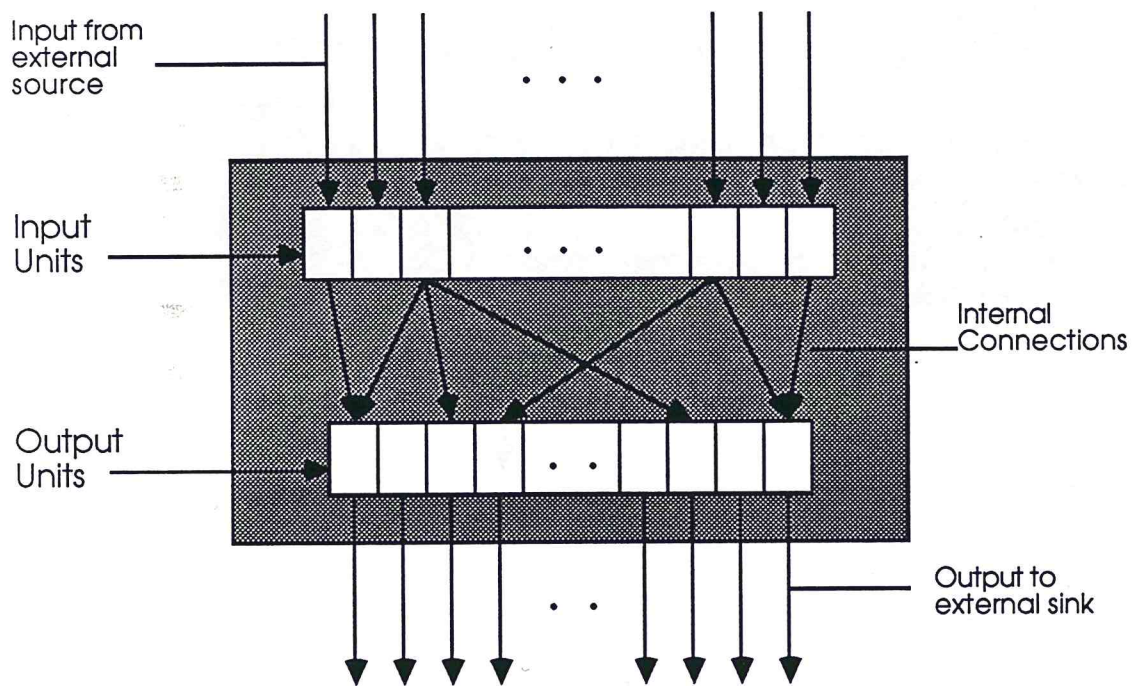


Figure 3 A Module with input and output units

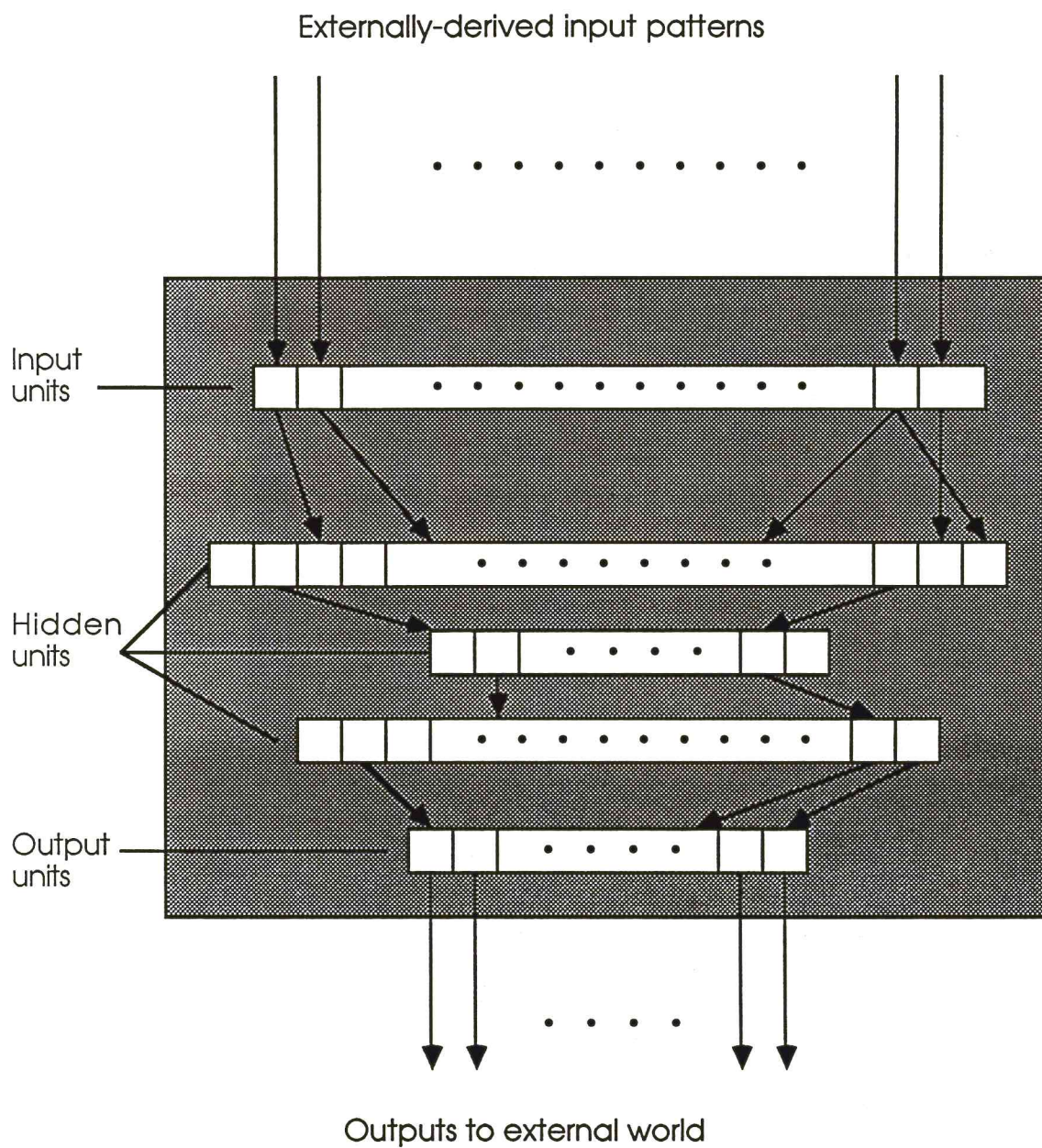


Figure 4 A Module with hidden units

(Note that layers need not all contain the same number of units.)

dimensional array of units. At present, facilities for using volumes are not included in modules, but users can load volumes into modules and perform all necessary operations explicitly using existing facilities. It is hoped that volumes will be better integrated with the rest of the package at a later date.

At present it is not possible to combine vectors and surfaces within a single WINNSOME module (although it is comparatively easy to do in theory). The effect of such combination can be achieved, though, by connecting a number of modules, each with the appropriate internal structure. For example, a module that computes the column-wise average intensity of an image can be constructed by inputting the image into a module containing a surface as its single layer. The output from this module is sent to another module that contains a vector of units. The units in each column of the surface held in the first module are connected to a single unit in the second. So, all the units in column one of the first module are connected to the first element of the vector in module two; those in column two are connected to the second element of module two's vector, and so on: this is shown in Figure 5.

3.2 Patterns and Activation Values: Module Input and Output

The internal structure of a module determines the kinds of input it can be given and the kind of output it can generate. Modules composed of vectors can accept and generate vectors of activation patterns; modules composed of surfaces can accept and generate surfaces of patterns. Patterns are sequences of *activation values*: these are abstract and uniform representations of the values presented to or generated by units. Activation values can be of any scalar type and type consistency is ensured by the simulator and by C++.

The existence of activation values means that the simulation builder need not be aware of the details of the units contained in a module: all that need be done is for the values to be supplied to the simulation and appropriate actions will be taken automatically.

Module containing a single surface - represents part of an image

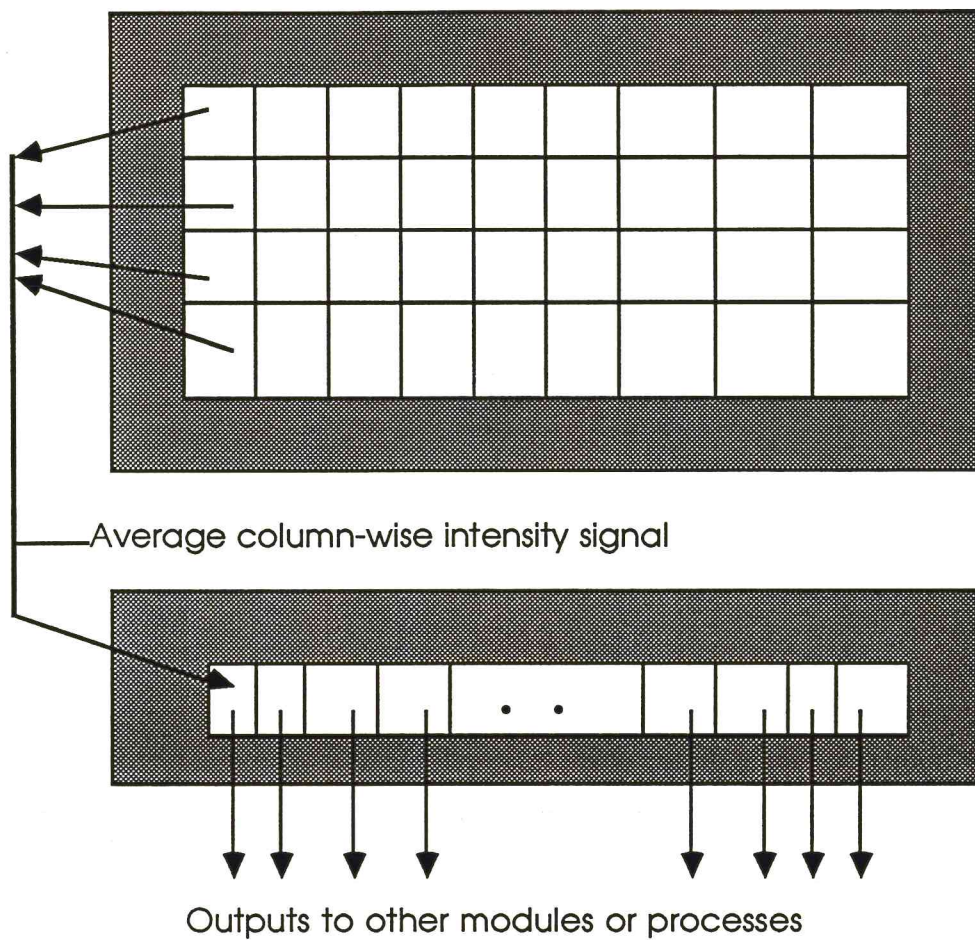


Figure 5 Two modules connected via an input/output relationship.

Input arrives at the top module and is passed to all units simultaneously. Activation is then passed to the single layer of the lower module.

Activation values provide a uniform method of transmitting information across a network of units. All of the basic constructs provided by WINNSOME operate using the activation value abstraction and do not directly access any pattern values. When a value has to be accessed, it is the job of the relevant unit or connection. There is sometimes a need to know what sort of value is represented by an activation value, particularly when extracting the value to perform some computation. This problem will be solved by appropriate network definition software that is still under construction. When passing patterns between units or between modules, though, the quantities represented by any particular activation value can be ignored. As has already been explained, there is a protocol that must be observed by all units and connections: this protocol is designed to ensure that the detailed workings of any unit or connection are independent of the simulator framework.

When a pattern is to be input to a module, a generic method, called `ACCEPT_INPUT` is called. This method checks to see that the pattern is of the correct shape (i.e., whether it is a vector or a surface) and that it has enough elements. If the pattern is appropriate, it is accepted and propagated to each of the module's input units (this is done by calling each unit's `SET_INPUT` method on the corresponding pattern element). Once a pattern has been input, it is up to the module to process it, propagate and transform it, eventually creating an output pattern. Output patterns can be propagated either to external sinks (e.g., statistical analysis packages, graphic interfaces) or to other modules. When an output pattern has been generated by a module, it can be accessed via the `GENERATE_OUTPUT` method that is defined for every module type. This method constructs an output pattern from the units that are held in the module's output vector.

In the simplest case, a simulation will contain only one module. Patterns are accepted from some external source (say a training-set file), and are processed by the units in the module. The results of this processing are then sent to some

external agent (e.g., a graphic display). However, since WINNSOME allows modules to be connected together in a variety of ways, patterns can also be sent from the generating module to some other module. This process can occur in a variety of ways and is described in section 3.4.

3.3 Propagation

As has been stated, propagation depends upon the structure of any particular model built using WINNSOME. The basic unit type provided by the package contains a propagation operation that can be executed on a per-unit basis: this operation is inherited by all units whether they are package- or user-defined. Because WINNSOME is intended to give the model builder as much freedom as possible, it also contains facilities to assist in writing propagation functions: these facilities are defined for each module type in the package.

It is possible to access each unit in a module: the module types have methods that permit this. Unit access can either be random (i.e., any unit can be accessed given its location in a module - this is directly analogous to random access memory), or it can be more structured. For example, modules that contain vectors of hidden units have methods that allow each vector to be accessed directly. The unit vector type has an iterator type associated with it so that the units in a vector can be accessed sequentially. Modules of this type also have a method which allows iteration over the hidden vectors so that each hidden layer can be accessed in turn. These methods can be used when implementing a special-purpose propagation function or when using the `PROPAGATE_ACTIVATION` method defined by default for every unit.

3.4 Inter-Module Connection

In the simplest scheme of things, a simulation only has one module. This module contains all the units and connections required to simulate the process under investigation. WINNSOME allows the user to build simulations composed of more

than one module. In such circumstances, the modules can take input from each other and direct output to modules as well as to external sinks.

The simplest inter-module connection is that in which one module, M_1 , say, generates output that is sent directly to another module, M_2 . Where M_1 derives its input from, and where M_2 sends its output are not at issue here. The point is that M_1 generates a pattern that is useful to M_2 which consumes it in some way. The output generated by M_1 must be such that it has the correct shape for input to M_2 . However, if M_1 's output is not of the correct shape, an additional module can be interposed between M_1 and M_2 to perform the appropriate transformation.

If, though, the output of M_1 is of the correct shape, there are two ways of connecting the modules. The first way is to use normal inter-unit connections, in which case, the relationship between the modules is just like that between layers in a normal multi-level module. The second way of passing the output pattern from M_1 to M_2 is to generate output in the normal way and to use the `ACCEPT_INPUT` method belonging to M_2 . The second scheme is shown in Figure 6. This way of passing activation has the disadvantage that transformations cannot be made to the pattern between units - if, though, one wanted that effect, it would be possible to introduce an additional module between M_1 and M_2 to perform the transformation.

This kind of inter-module connection allows modules to be chained together: the output of one module serves as the input of another. WINNSOME supports other connection schemes in an attempt to be as general as possible. Basically, the only limit on the inter-module connectivity is the requirement of any particular simulation. However, an extra kind of inter-module connection will now be described so that the reader gains an idea of the kind of schemes that are possible. The other kind of inter-module connection was suggested by the projections from deep brain regions to the neocortex.

Imagine a pair of modules, M_1 and M_2 . M_1 obtains input from somewhere and derives output via its output units. Similarly, M_2 takes input from somewhere and

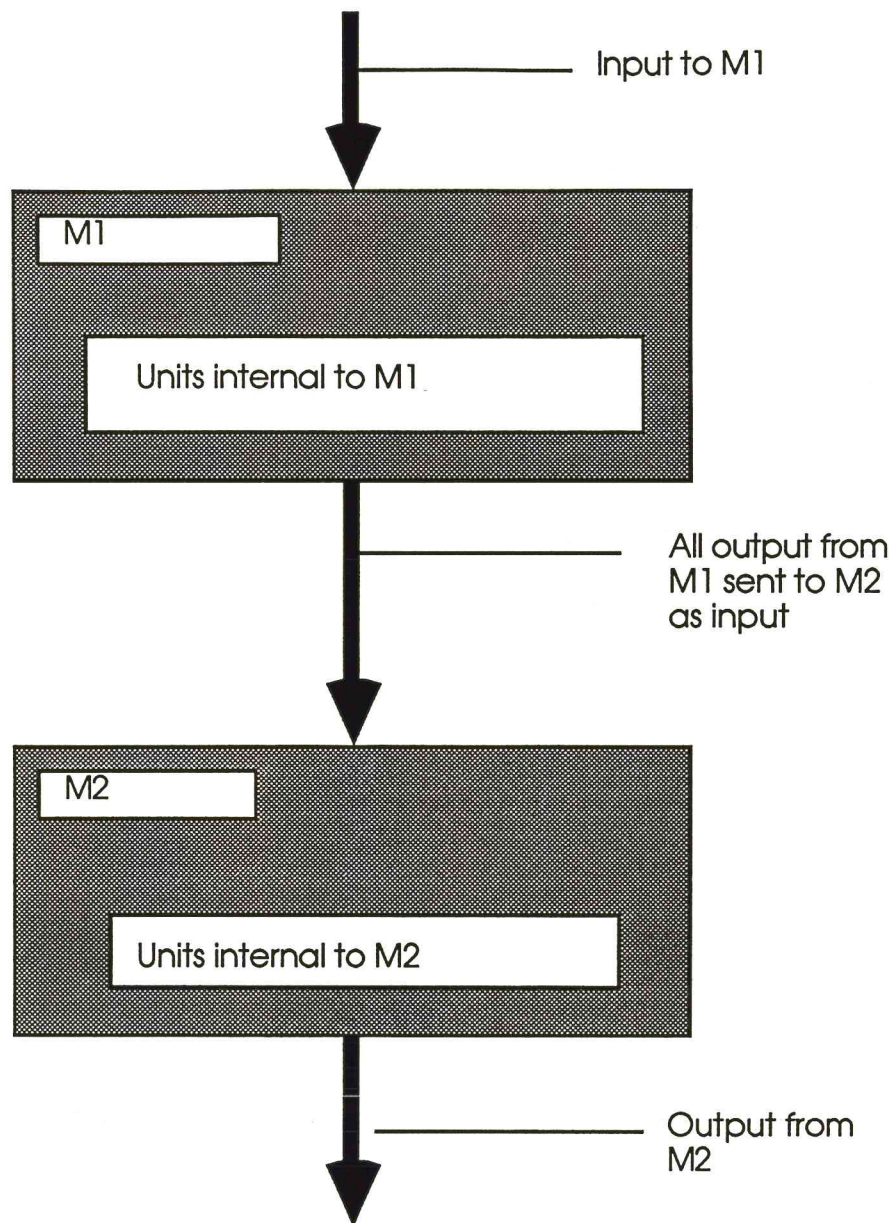


Figure 6 Input/output relationship between two modules

also generates output. However, some units in M_1 are connected *directly* to units inside M_2 . In other words M_1 projects to M_2 , as is depicted in Figure 7. There are no restrictions imposed by the WINNSOME software on which units can be connected to units in other modules: this is exemplified by the current example.

What happens in this example system is that M_1 and M_2 receive input patterns and can begin processing. Assuming a sequential activation algorithm, at some point, one or more of M_2 's units will update their state and this will involve obtaining the activation values from the units in M_1 to which they are connected.

As far as the WINNSOME software is concerned, there is no difference between a connection between two units that are inside the same module and units in different modules, so there is no overhead involved in accessing M_1 's units from inside of M_2 . The reader is warned that parallel or psuedo-parallel versions of the simulator might make inter-module access more costly, though. The output of M_2 will depend, because it is connected to units in M_1 , upon the state of M_1 . The ways in which M_2 's output will be affected are not known at the time of writing because they will tend to be simulation- and function-specific.

The point of the above description is that WINNSOME allows such connections to be made and exploited. WINNSOME also allows units inside a module to be connected to the input units of another. Similarly, the output units of a module can be connected directly to inputs inside another module. These facilities allow WINNSOME simulations to have a very rich interconnection structure. This structure can also be very flexible because it is possible to connect a set of units of different types to a single unit in the same or in another module - such a unit might function rather like a Sigma-Pi unit insofar as it summarises the state information of its input units - the output from this summarising unit can be passed to one or more units at various points in the simulation.

3.4 Modules and Architecture

The inter-module connection facilities provided by WINNSOME can be used to

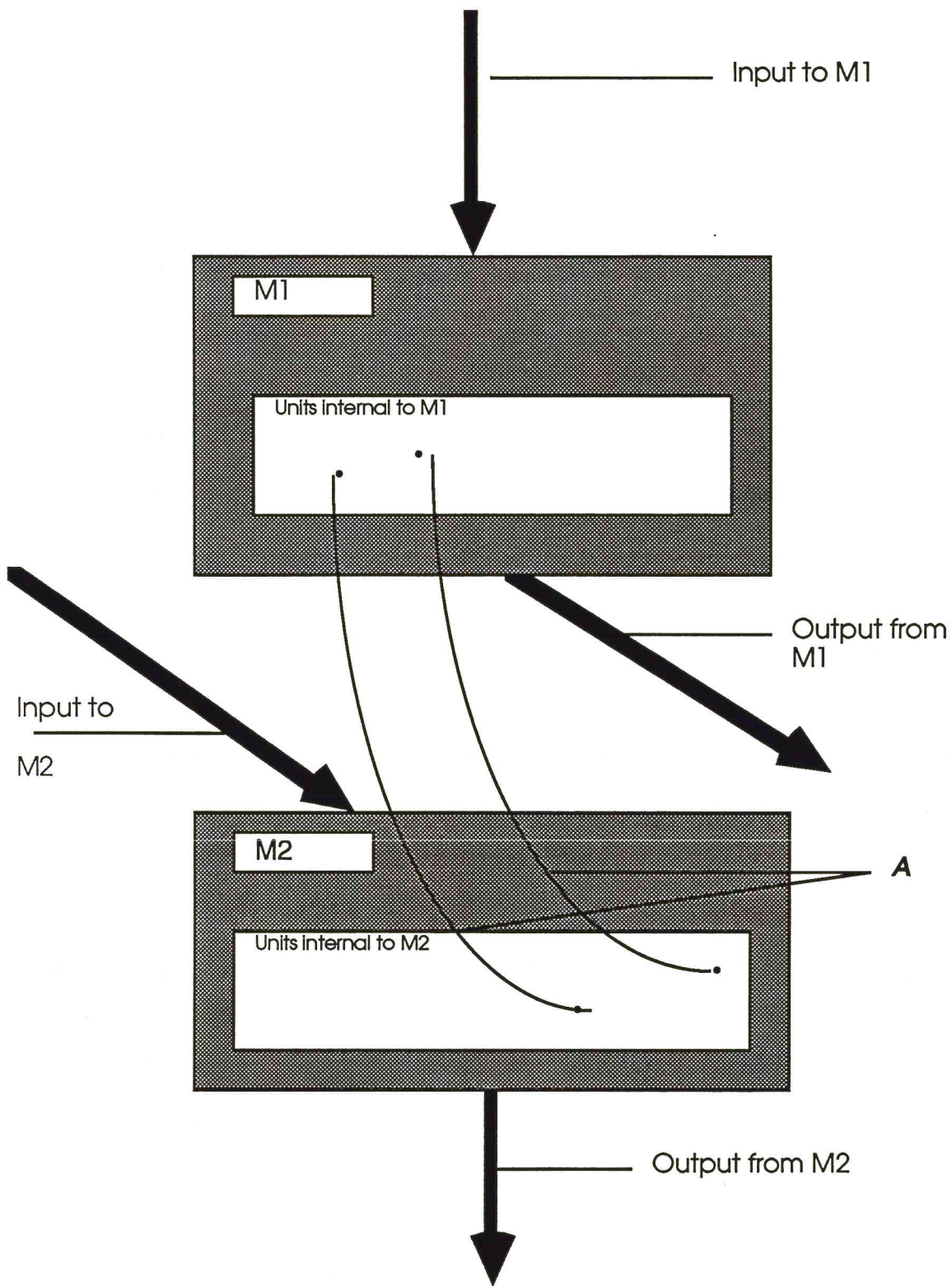


Figure 7 Direct connection between two modules. Bold italic ***A*** denotes the direct connections

model information-carrying pathways such as those found in a number of brain regions. Because units inside modules can be connected to units inside other modules, the idea of representing a pathway as a separate module should be clear. Indeed, the requirement that it should be possible to develop such structures partially motivated the module construct in the first place.

Modules were initially suggested by the macro-columns of the neocortex, but additional reasons were soon found for including them as distinct objects in the simulation package. One particularly important justification of the module construct is that it allows simulations to be constructed in a modular fashion. Thus, if it is known that a simulation is to include a number of different functions, each function can be represented in terms of a separate module. The module construct also allows input-output relationships to be described fairly readily: so, if there are five functions that are related to each other, the inputs and the outputs can be defined in terms of the inputs and outputs of the five modules that implement those functions.

A problem with some simulation software (e.g., P3) is that it does not allow the modeller to divide the simulation into neat pieces that can be tested independently. Since modules can be constructed and tested independently of each other, the construction process should be eased somewhat. Another problem with less structured networks is that identifiable functions cannot be neatly separated. Again, WINNSOME avoids this problem. However, it was also recognised that it would not be acceptable to represent modules in a way that was too much like a black box. This is the reason for allowing connections between units inside modules; the neurophysiological evidence for allowing such freedom has already been cited and seems compelling.

Since WINNSOME is still under development and because large-scale simulations are only beginning to be written, it is hard to describe the practical advantages of modules. Theoretically, they appear to have a firm basis, but the actual use of them is only just starting. Perhaps it will be possible, for example, to

construct libraries of standard modules that can be included in any simulation at will; perhaps it will be the case that, until we have a far better understanding of the theory of neural networks, each module will have to be constructed afresh for each new simulation study.

4 CONCLUSIONS

This paper has briefly described and motivated the WINNSOME neural network simulation package. The package was designed with two aims in mind:

- to allow users to define new unit and connection types according to the needs of their problem, and
- to allow units to be collected together into modules so that complex, modular networks can be constructed.

WINNSOME is written in an object-oriented programming language and facilities provided by the language are exploited by the package. In particular, the inheritance mechanism provided by object-oriented languages has been used to allow heterogeneous networks to be constructed and to provide protocols to ease the introduction of new types into the package.

The programming language in which the package is implemented was chosen because of its type structure, but also because it compiles into fairly fast code. The execution speed of simulation code was considered to be particularly important because it is anticipated that some large models will be built during the next few years - anything which eases the problem of getting results in adequate time was considered to be worth investigating.

The major developments of the package are the module construct and the ability to mix units and connections of different types. The module construct was introduced so that groups of units could be collected into identifiable and independent regions of the simulation. Modules also allow interfaces between the components of a simulation to be defined and enforced. The need to mix units of

different types has already been explained, but it should be pointed out again that modules can contain units of any type whatsoever.

The package is still under development and large-scale models have yet to be built. The network definition and graphic interface components are still under construction at the time of writing. In addition to these facilities for defining and monitoring simulations, the package will be altered so that it can support concurrent and pseudo-concurrent execution. Precisely how concurrency can be introduced in WINNSOME simulations and how the simulator structure will have to be changed (if at all) are still open questions, but the option of compiling simulations down into structures that can be directly executed on the Inmos Transputer™ is being carefully considered.

It is hoped that with experience, the facilities provided by the package will expand so that an increasing number of operations can be included automatically in simulations.

REFERENCES

- Rumelhart (1986) Rumelhart, D.E., McClelland, J.L. et al., *Parallel Distributed Processing*, MIT Press, Cambridge, MA, 1986.
- Shepherd (1979) Shepherd, G.M., *The Synaptic Organization of the Brain*, Oxford University Press, New York, 1969.
- Skarda (1987) Skarda, C.A. and Freeman, W.J., How brains make chaos in order to make sense of the world, *Behavioral and Brain Sciences*, Vol. 10, pp. 161 - 195, 1987.
- Soustrup (1986) Soustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.
- Squire (1987) Squire, L.R., *Memory and Brain*, Oxford University Press, New York, 1987.
- Zipser (1986) Zipser, D. and Rabin, D.E., P3: A Parallel Network Simulating System, *in* Rumelhart (1986), Vol. 1, pp. 488 - 506.

